Separate Compilation of Bayesian Networks for Efficient Exact Inference

Shan Gao
Graduate School of Information Science and Technology Hokkaido University
gao.shan@ist.hokudai.ac.jp

Masakazu Ishihata
(Ditto)
ishihata.masakazu@ist.hokudai.ac.jp

Shin-ichi Minato
(Ditto)
minato@ist.hokudai.ac.jp

keywords: Bayesian Network, exact inference, Zero-suppressed BDD, $d$-separation, separate compilation

Summary

Compiling Bayesian Networks (BNs) into Zero-Suppressed BDDs (ZDDs) to perform efficient exact inference has attracted much attention. Computation time for exact inference using ZDDs is reduced to linear time in the size of the ZDDs. Also, cache memory techniques further help accelerate the inference. However, as the size of BN grows, compiling ZDDs becomes unacceptable in both time consumption and ZDD size which hinders BN practical applications. In this paper, we aim to improve the conventional ZDD-based method by proposing the idea of partitioning and separately compiling BNs. For every given BN, serial pattern $d$-separation sets are found and used to partition the BN into conditionally independent components. Separately compiling these components into ZDDs is more efficient than generating a giant ZDD for a whole network. However, partitioning a BN into too many components may give rise to considerable time consumption which grows exponentially with the number of vertexes in serial pattern $d$-separations. To trade off the off-line time consumption (for finding $d$-separations and compiling ZDDs) and on-line time consumption (for inference using ZDDs), the $d$-separations used to partitioning BNs are restricted to one-vertex and found using Tarjan’s vertex-cut algorithm which can be performed linear time in the number of BN vertexes. The experiments illustrate that one-vertex $d$-separations exist in most BNs. Partitioning BNs with one-vertex $d$-separations improves the speed for both compilation and inference largely than the conventional ZDD-based method. To show the validity of partitioning with one-vertex $d$-separations, we also conduct the experiments of partitioning with two-vertex $d$-separations and the comparative experiments of jointree algorithms.

1. Introduction

A Bayesian Network (BN) [Pearl 88] is a probabilistic graphical model that represents a probability distribution over a set of random variables. The Exact inference in a BN is to compute the probability distribution over the variables of interest and known as NP-hard [Cooper 88, Cooper 90, Koller 09]. Approaches of compiling BNs into structured representations such as, Zero-suppressed BDDs (ZDDs), Affine Algebraic DDs (AADDs), Probabilistic Sentential Decision Diagrams (PSDD) [Kisa 14, Minato 93, Scott 05] have been widely used for efficient exact inference in BNs. These compilation approaches yield significant time and space savings over the conventional tabular representations [Jensen 96] on probability computations. 

In this paper, we improve the conventional ZDD-based compilation approach [Minato 07] by a new divide-and-conquer method which reduces the time consumption for both compilation and exact inference.

A BN can be characterized by a Multi-Linear Function (MLF) [Darwiche 00]: a polynomial combination of propositional variable sets corresponding to realizations of random variables. The conventional ZDD-based method compiles the MLF into ZDDs to compactly represent the BN. Each ZDD represents the MLF of each BN vertex. The ZDD for a BN vertex depends only on the ZDDs of its ancestor vertexes. Exact inference is carried out in a time almost linear in the ZDD size based on the multi-value multiplication algorithm of ZDDs [Minato 07]. Node sharing techniques of ZDDs help exploit local structures such as determinism [Chavira 05, Friedman 96] and local computation [Gao 17, Lauritzen 88]. Cache memory techniques
of ZDDs further accelerate the inference significantly [Minato 07]. However, since the size of MLFs is exponential to the number of variables, when the BN gets large, the size of ZDDs becomes unacceptable and considerable time for compiling a BN into ZDDs puts limits to the usefulness of the conventional ZDD-based compilation approach.

Recently, Gao, et al. [Gao 15] has proposed a method to improve the conventional ZDD-based method which can condense the ZDD size through factorizing a large ZDD into several small ones [Geiger 90] of BNs. Their method is based on a large ZDD representing the whole BN using the conventional ZDD-based method. They manually find d-separation vertexes and generate ZDDs for these d-separations and then use them as divisors to factorize ZDDs of the whole BN. Through their method, large ZDDs can be factored into small ones efficiently using the fast weak division algorithm [Gao 15]. But the original ZDD for the whole BN is too large and too time consuming to generate using the conventional ZDD-based method. Also, the d-separations used are found manually in an ad-hoc way. Moreover, they only compare the ZDD size with the conventional ZDD-based method but did not conduct experiments for exact inference, as they assume that computation time of inference is linear to the ZDD size.

In this paper, to improve the conventional ZDD-based method, we propose an idea of first partitioning a given BN into several conditionally independent components using serial pattern d-separations. Then, these components are separately compiled into ZDDs. Though there may be many serial pattern d-separations in a given BN, partitioning a BN into too many components would slow down the inference. To trade off the time between compilation and inference, the serial pattern d-separations are restricted to one-vertex and found using Tarjan’s vertex-cut algorithm [Tarjan 74] which costs linear time in the number of BN vertexes. Comparing with the conventional ZDD-based method, the total ZDD size and time for compilation are largely reduced. The efficiency of inference using ZDDs is guaranteed by restricting the d-separation size to one vertex. To show the validity of partitioning with one-vertex d-separations, we also conduct the experiments of partitioning with two-vertex d-separations. The results show that one-vertex d-separations exist in most networks and partitioning with them works better than partitioning with two-vertex d-separations while performing on-line inference.

The paper is structured as follows: We describe the basic concept of BN compilation and the conventional ZDD-based method in Chapter 2. In Chapter 3, we first introduce our idea of partitioning BNs using serial pattern d-separations. Then we discuss the algorithms to find all one-vertex serial pattern d-separations using Tarjan’s vertex-cut algorithm. In Chapter 4, we show the details of how to compile a BN into ZDDs after the partitioning and introduce on-line inference with these ZDDs. Finally, we present the experiment results compared with the conventional ZDD-based method and related works in Chapter 5 and 6.

2. Preliminaries

2.1 Bayesian Networks and MLFs

A Bayesian Network (BN) [Pearl 88] is a Directed Acyclic Graph (DAG) which defines a joint distribution over a set of random variables. A BN is defined as $B \triangleq \langle G, \Theta \rangle$, where $G \triangleq \langle V, A \rangle$ is a DAG that each vertex represents a random variable and edges between vertexes represent the dependencies among corresponding variables. We use $V \triangleq \{X_1, \ldots, X_n\}$ to represent all the vertexes and $A \subset V \times V$ as the set of directed arcs that represent dependencies between vertexes. The set of parents of $X_i$ in a BN is defined as $\Pi_i \triangleq \{X_j \in V \mid (X_j, X_i) \in A\}$. The second parameter $\Theta \triangleq \{\theta_{i,j,k}\}_{i,j,k}$ denotes a set of conditional probabilities of vertexes where $\theta_{i,j,k}$ indicates the probability of $X_i$ taking the k-th value $x_{i,k}$ given its set of parents $\Pi_i$ taking the j-th instantiation $\pi_{i,j}$: $\theta_{i,j,k} \triangleq P(X_i = x_{i,k} \mid \Pi_i = \pi_{i,j})$. A BN assumes that the conditional probability distribution (CPD) at each vertex depends only on its parents. Thus, given a BN $B$, the joint distribution of vertexes in $V$ defined by $B$ is represented as:

$$P(X_1, X_2, \ldots, X_n) = \prod_i^n P(X_i \mid \Pi_i).$$  \hspace{1cm} (1)

A BN which consists of vertexes $\{X_1, X_2, X_3, X_4\}$ is shown in Figure 1. The CPD for every vertex is presented in tables known as Conditional Probability Tables (CPTs). Parameters such as $\theta_{4,1,1}$ for $X_4$ mean that $X_4$ takes its first value $x_{4,1}$ with its parents set $\{X_2, X_3\}$ instantiated with the first instantiation $\{(x_{2,1}, x_{3,1})\}$. Similarly, $\theta_{4,4,2}$ means that $X_4$ takes the second value $x_{4,2}$ with its parents instantiated with the fourth instantiation $\{(x_{2,2}, x_{3,2})\}$.

In this paper, boldface letter $X \subset V$ indicates a set of vertexes. Its domain is referred as Dom$(X) = \{x_l\}$ where $x_l$ is the l-th instantiation of $X$ and $l \in \mathbb{N}^+$ ranges from 1 to the total number of instantiations of variables in $X$. For the BN in Figure 1, for each vertex, we have Dom$(\{X_1\}) = \{x_{1,1}, x_{1,2}\}$, Dom$(\{X_2\}) = \{x_{2,1}, x_{2,2}\}$, Dom$(\{X_3\}) = \{x_{3,1}, x_{3,2}\}$, Dom$(\{X_4\}) = \{x_{4,1}, x_{4,2}\}$. Also, for any
given vertex set such as \( \{X_2, X_3\} \), \( \text{Dom}(\{X_2, X_3\}) = \{(x_{21}, x_{31}), (x_{22}, x_{31}), (x_{21}, x_{32}), (x_{22}, x_{32})\} \). The joint distribution for vertexes in \( X \) is represented as \( P(X) \). 

\( P(x_i) \) indicates the probability of \( X \) instantiated with \( x_i \). Usually, we write \( P(x_i) \) for short. We use the notation \( \text{Ances}(X) \) to represent all the ancestor vertexes of \( X \) in the BN: \( \text{Ances}(X) = \{X_i \mid \text{at least one of the vertexes in } X \text{ is reachable from } X_i\} \). Thus, we have \( \text{Ances}(\{X_i\}) = \{X_1, X_2, X_3\} \).

Given a BN representing a joint distribution over all the vertexes, marginal probability of any given vertex set \( X \subset V \) instantiated with \( x_i \) can be calculated by summing over all possible instantiations of all other vertexes:

\[
P(x_i) = \sum_{V \setminus X} P(x_1, \ldots, X) \text{,} \quad \text{where } V \setminus X \text{ means the complement of } X \text{ in } V. \tag{2}
\]

Usually, \( \sum_{X_i} \) refers to summing over all possible values that \( X_i \) can take. We use notation \( \sum_{X} \) as a shorthand for \( \sum_{X_1} \sum_{X_2} \cdots \sum_{X_4} \), summing over all vertexes in \( X = \{X_1, X_2, \ldots, X_4\} \).

For example, the probability of \( P(x_{4,2}) \) in Figure 1 can be computed as:

\[
\sum_{x_1, x_2, x_3} P(x_1) P(x_2 | x_1) P(x_3 | x_1) P(x_4 = x_{4,2} | x_2, x_3). \tag{3}
\]

Note that even in the simplest case that every vertex is binary-valued, computation time for probability calculation of any given vertex sets \( X \) grows exponentially with the number of vertexes in a BN. Such computation is usually prohibitive in exact inference. One direct method for improve the computation is to use Multi-Linear Functions (MLFs) [Darwiche 00]. An MLF represents the joint distribution induced by a BN. An MLF consists of two types of variables, i.e., an indicator variable \( \lambda_{i,k} \in \{0, 1\} \) and a parameter variable \( \theta_{i,j,k} \). \( \lambda_{i,k} = 1 \) means that vertex \( X_i \) takes its \( k \)-th value and \( \lambda_{i,k} = 0 \) otherwise. An MLF contains terms for all instantiations of vertexes:

\[
\text{MLF}_V = \sum_{l: x_l \in \text{Dom}(V)} \prod_{i,j,k} \lambda_{i,k} \theta_{i,j,k}. \tag{4}
\]

\( \prod_{i,j,k} \) refers to the multiplication of variables ranging with all values that \( i, j, k \) can take. Likewise, \( \sum_l \) refers to the summation over variables ranging with all values that \( l \) can take. The MLF for the example in Figure 1 is given by:

\[
\text{MLF}_V = \lambda_{1,1} \lambda_{1,2} \lambda_{1,3} \lambda_{2,1} \lambda_{3,1} \lambda_{2,1} \lambda_{4,1,1} \lambda_{2,1} \lambda_{3,1} \lambda_{4,1,1} + \lambda_{1,2} \lambda_{2,2} \lambda_{3,2} \lambda_{4,2,1} \lambda_{3,2} \lambda_{2,2} \lambda_{4,2,1} + \lambda_{1,3} \lambda_{3,3} \lambda_{4,3,1} \lambda_{3,3} \lambda_{2,3} \lambda_{4,3,1} + \lambda_{2,3} \lambda_{3,4} \lambda_{4,4,1} \lambda_{3,4} \lambda_{2,4} \lambda_{4,4,1} + \lambda_{2,4} \lambda_{3,4} \lambda_{4,2,2} \lambda_{4,2,2} \lambda_{4,4,2} \lambda_{3,4} \lambda_{2,4} \lambda_{4,4,2}. \tag{5}
\]

Computing the probability of an instantiation of any vertexes using MLFs is performed by setting all indicators variables consistent with the instantiation to 1 and otherwise to 0. For example, the marginal probability of \( x_{4,2} \) is computed by evaluating this MLF through respectively setting \( \lambda_{4,1} \leftarrow 0, \lambda_{4,2} \leftarrow 1 \) (other \( \lambda s \) are set to 1 and \( \theta s \) are set to the values in CPT). Time of exact inference using this MLF is linear in the numbers of variables in the MLF [Darwiche 00]. Because the number of variables in the MLF grows exponentially with the number of vertexes in the BN, the exact inference by a MLF is quite time-consuming. However, if the MLF is factored into a compact arithmetic expression, it is possible to speed up the inference. One way to do factorization using Zero-suppressed BDDs has been proposed in [Minato 07].

### 2-2 Zero-suppressed BDDs

A Zero-suppressed Binary Decision Diagram (ZDD) [Minato 93] is a compact DAG representation of a Boolean function. A ZDD consists of two terminal nodes\(^*\), 0-terminal node and 1-terminal node, and many decision nodes with exactly two outgoing edges, called 0-edge and 1-edge. Each decision node is labeled by a Boolean variable and 1-edge (0-edge) indicates that the variable is true (false). A ZDD package supports a set of various basic logic operations (i.e., AND, OR and XOR) for given a pair of operand ZDDs. Its computation time is almost linear in the ZDD size. By using those inter-ZDD operations, one can construct ZDDs for any given Boolean functions. The details of ZDDs are described in [Minato 93].

\(^*\) We use the terms "nodes" and "edges" in ZDDs to distinguish with the terms "vertexes" and "arcs" in BNs.
ZDDs are efficient representations of not only Boolean functions but also combinatorial itemsets [Minato 01]. Given $m$ items, an itemset is a subset of items. Thus, there are $2^m$ possible itemsets. A combinatorial itemset is a family of itemsets; there are $2^m$ possible combinatorial itemsets. For example, given five items $a, b, c, d,$ and $e$, examples of combinatorial itemsets include:

$$\{ab, c\}, \{abc, bd, acde, e\}, \{1, cd\}, \emptyset,$$  \hspace{1cm} (6)

where “1” denotes the empty itemset and $\emptyset$ denotes the empty combinatorial itemset. A combinatorial itemset can be represented as a Boolean space of $m$ input binary variables. For example, the truth table of the Boolean function $F = (a\overline{b}c) \lor (b\overline{c})$ in Figure 2(a) also represents the combinatorial itemset $S = \{ab, ac, c\}$, which is the family of input itemsets that makes $F$ true. An example of ZDD representing $S$ is shown in Figure 2(b). A path in the ZDD from the root node to the 1-terminal node corresponds to an itemset of $S$. Nodes of irrelevent items (never appeared in this itemset) are automatically deleted from the path.

Given multiple combinatorial itemsets of the same items, isomorphic subgraphs of their ZDDs can be shared under the same fixed input binary variable order. Figure 3 shows an example of the shared ZDDs for combinatorial itemsets $I$ and $S$. Using node-sharing techniques, one can handle a number of combinatorial itemsets in the time approximately proportional to the compressed ZDD size but not the number of terms of the combinatorial itemset.

An MLF is a polynomial in the indicator and parameter variables. Since each term of MLF is simply a combination of variables, it can be represented compactly by a ZDD.

2-3 Compiling BNs into ZDDs

The conventional ZDD-based method compiles each BN vertex into one MLF based on the MLFs of its parents. Multiplying all the MLFs can get the MLF representing the joint distribution of the given BN equivalent to equation (4). For each BN vertex $X_i$, the MLF is recursively defined as:

$$\text{MLF}_{X_i} = \sum_{k, x_{i,k} \in \text{Dom}(X_i)} \text{MLF}_{X_{i,k}},$$  \hspace{1cm} (7)

$$\text{MLF}_{x_{i,k}} = \lambda_{i,k} \sum_{j, \pi_{i,j} \in \text{Dom}(\Pi_i)} (\theta_{i,j,k} \prod_{i', k' : x_{i',k'} \in \pi_{i,j}} \text{MLF}_{x_{i',k'}}).$$  \hspace{1cm} (8)

For the vertex with no parents,

$$\text{MLF}_{x_{i,k}} = \lambda_{i,k} \theta_{1,1,1}, \text{where } \Pi_i = \emptyset.$$  \hspace{1cm} (9)

For the example in Figure 1, the MLF at vertex $X_1$ is:

$$\text{MLF}_{X_1} = \text{MLF}_{x_{1,1}} + \text{MLF}_{x_{1,2}} = \lambda_{1,1} \theta_{1,1,1} + \lambda_{1,2} \theta_{1,1,2}.$$  \hspace{1cm} (10)

Then the MLF for $X_2$ can be written as:

$$\text{MLF}_{X_2} = \lambda_{2,1} (\theta_{2,1,1} \text{MLF}_{x_{1,1}} + \theta_{2,2,1} \text{MLF}_{x_{1,2}})$$
$$+ \lambda_{2,2} (\theta_{2,1,2} \text{MLF}_{x_{1,1}} + \theta_{2,2,2} \text{MLF}_{x_{1,2}})$$
$$= \lambda_{1,1} \lambda_{2,1} \theta_{1,1,1} \theta_{2,1,1} + \lambda_{1,1} \lambda_{2,2} \theta_{1,1,1} \theta_{2,1,2}$$
$$+ \lambda_{1,2} \lambda_{2,1} \theta_{1,1,2} \theta_{2,2,1} + \lambda_{1,2} \lambda_{2,2} \theta_{1,1,2} \theta_{2,2,2}.$$  \hspace{1cm} (11)

A ZDD for each MLF is constructed by the multi-valued multiplication algorithm in [Minato 07]. This algorithm generates ZDDs for the multiplications of two MLFs which contain all possible combinations of terms from the respective MLFs. Variables such as $\lambda_{i,k}$ and $\lambda_{i,k'}$ ($k \neq k'$) (variables representing different instance of the same vertex) do not coexist in the same term as they are mutually exclusive. Also, the multi-valued multiplication algorithm with ZDD operations makes sure that no term can contain the same variable more than once, so instead of $\lambda_{i,k}^2$ for duplicate variables, simply $\lambda_{i,k}$ will appear in the result. An implicit factored representation of MLF $\text{MLF}_{X_i}$ in equation (11) with node sharing ZDDs is presented in Figure 4. The
ZDD for MLF\(_{X_2}\) containing only variables relevant to \(X_2\) and its ancestor \(X_1\). In this example, there are four paths from the root node to the 1-terminal node, each of which corresponds to a term of MLF\(_{X_2}\).

While inferring using ZDDs, to calculate the probability of any given instance \(x_i\) over vertex set \(X\), we first generate a ZDD of MLF\(_{x_i} = \prod_{k : x_k \in \mathbb{X}}\text{MLF}_{x_k, X}\) by multiplying ZDDs of every MLF\(_{x_k, X}\) using the multi-valued multiplication algorithm. Contradicting terms are automatically eliminated, so the result of ZDD contains only the variables related to the query. We then set all indicators to 1 and trace all ZDD paths ending with 1-terminal node to calculate the probability. Comparing with equation (2), computing marginal probability for any given instances using the conventional ZDD-based method is reduced to the sum of joint distributions over their ancestor vertices:

\[
P(x_i) = \sum_{\text{Ances}(X)} P(X = x_i, \text{Ances}(X)).
\]

Irrelevant variables of non-descendants are out of consideration automatically.

For the example of the BN in Figure 1, to calculate the probability of \(P(x_{2,2}, x_{3,1})\), we have:

\[
\text{MLF}_{x_{2,2}} = \lambda_{1,1}\lambda_{2,2}\theta_{1,1,1}\theta_{2,1,2} + \lambda_{1,2}\lambda_{2,2}\theta_{1,1,2}\theta_{2,2,2}.
\]

\[
\text{MLF}_{x_{3,1}} = \lambda_{1,1}\lambda_{3,1}\theta_{1,1,1}\theta_{3,1,1} + \lambda_{1,2}\lambda_{3,1}\theta_{1,1,2}\theta_{3,2,2}.
\]

We generate:

\[
\text{MLF}_{x_{2,2}}\times\text{MLF}_{x_{3,1}} = \lambda_{1,1}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,1}\theta_{2,1,2}\theta_{3,1,1} + \lambda_{1,2}\lambda_{2,2}\lambda_{3,1}\theta_{1,1,2}\theta_{2,2,2}\theta_{3,2,2}.
\]

By setting all \(\lambda\)s to 1, finally we can get the probability is \(0.4 \times 0.2 \times 0.5 + 0.6 \times 0 \times 0.5 = 0.04\).

Using ZDDs, the MLF for any given BN is compactly represented. However, when the number of vertexes in \(\text{Ances}(X)\) is too large, time for compilation and computing \(P(X = x_i, \text{Ances}(X))\) in equation (12) would be extremely time consuming. Therefore, methods to reduce time for both compilation and inference need to be put forward.

3. Proposed Method

We propose a new ZDD-based compilation of partitioning a BN into conditionally independent components using serial pattern \(d\)-separations and then separately compiling these components into ZDDs. To trade off time consumption for compiling and inference, the proposed method restricts the size of \(d\)-separation to one vertex and use them all to partition a BN. Through partitioning BNs as a set of sub-problems, not only we can expect to reduce time consumption for off-line compilation, but also to improve on-line inference efficiency.

3.1 Partitioning BNs using \(d\)-separations

To resolve the problem of taking too long to constructing ZDDs for a large BN, naturally we think of partitioning the BN into a set of small components.

A \(d\)-separation[Geiger 90] is a vertex set which is usually used to check conditional independence among vertexes in BN structure learning[Heckerman 98]. Let \(X_1\), \(X_2\) and \(X^d\) be three vertexes. Then their dependence is categorized in three graph patterns shown in Figure 5, the serial pattern, diverging pattern and converging pattern.

If the three vertexes satisfy serial pattern \(d\)-separation and diverging pattern \(d\)-separation as shown in Figure 5, we say \(X^d\) is a \(d\)-separation of \(X_1\) and \(X_2\). This implies that if \(d\)-separation \(X^d\) is given, \(X_1\) and \(X_2\) become independent to each other (The converging pattern is normally treated as a \(d\)-connection: \(X_1\) and \(X_2\) becomes dependent if \(X^d\) is given).

The independence property prompts us the idea of partitioning a BN using \(d\)-separation vertex sets, that is, finding a vertex set \(X^d\) that partitions the BN vertexes into
two disjoint vertex sets $X^{(1)}$ and $X^{(2)}$ separately. We use $X^{(m)}$ to indicate different vertex set with different $m \in \mathbb{N}^+$. If $X^{(1)}$, $X^{(2)}$ and $X^d$ satisfy the $d$-separation patterns of (a) and (b), vertexes in $X^{(1)}$ is independent to vertexes in $X^{(2)}$ given $X^d$. Then, the BN can be partitioned into two components as shown in Figure 6.

Figure 6 partitioning with $d$-separations.

[Definition 1] Partition a BN: Given a BN $G = (V, A)$, find a vertex set $X^d$ to partition it into two components $G^1 = (X^{(1)} \cup X^d, A^1), G^2 = (X^{(2)} \cup X^d, A^2)$ such that:

- $X^{(1)} \cap X^{(2)} = \emptyset, X^{(1)} \cup X^{(2)} \cup X^d = V$.
- $X^d$ $d$-separates $X^{(1)}$ and $X^{(2)}$.
- $\forall X_i \in X^{(1)}, \forall X_j \in X^{(2)}$, $X_i$ is a non-descendant of $X_j$.
- $A^1 = \{(X_i, X_j) \mid (X_i, X_j) \in A, X_i, X_j \in V^1\}$, $A^2 = \{(X_i, X_j) \mid (X_i, X_j) \in A, X_i, X_j \in V^2\}$, where $V^1 = X^{(1)} \cup X^d, V^2 = X^{(2)} \cup X^d$.

First, we consider the partitioning with serial pattern $d$-separation. After the partitioning shown in Figure 6(a), the joint distribution of this BN is factorized as:

$$P(X^{(1)}, X^d, X^{(2)}) = P(X^{(2)} \mid X^d) P(X^d \mid X^{(1)}) P(X^{(1)})$$

$$= P(X^{(2)} \mid X^d) P(X^{(1)}, X^d).$$

(16)

Our method separately compiles the MLFs for $P(X^{(1)}, X^d)$ and $P(X^{(2)} \mid X^d)$ into ZDDs. Marginal probability such as $P(X^{(2)} = x_i^{(2)})$ can be computed by summing out irrelevant vertex set $X^{(1)}$ in advance:

$$P(x_i^{(2)}) = \sum_{X^d} P(X^{(2)} = x_i^{(2)} \mid X^d) \sum_{X^{(1)}} P(X^{(1)}, X^d).$$

(17)

In the conventional ZDD-based method, it compiles $P(X^{(1)}, X^d, X^{(2)})$ into one large ZDD (according to equation (12)) and computes:

$$P(x_i^{(2)} = \sum_{X^{(1)} \cup X^d} P(X^{(1)}, X^d, X^{(2)} = x_i^{(2)}).$$

(18)

Comparing with the conventional ZDD-based method, first, our method reduces time consumption of compiling $P(X^{(1)}, X^d, X^{(2)})$ by separately compiling $P(X^{(2)} \mid X^d)$ and $P(X^{(1)}, X^d)$. Secondly, the computation for $P(x_i^{(2)})$ are accelerated by summing out irrelevant vertexes $X^{(1)}$ in advance. Thirdly, if the on-line inference is only about vertexes $X^{(2)} \subset X^{(2)}$ which is computed using the probability $P(X^d)$, it can be accelerated through this partitioning by first computing and storing $P(X^d)$ off-line.

Next, we consider the partitioning with diverging pattern $d$-separation. Partitioning in Figure 6(b) means that if we partition this BN with vertex set $X^d$, its descendant vertexes in different branch with $X^d$ form different components. While compiling them into ZDDs, we construct ZDDs for $X^{(1)}$ and $X^{(2)}$ separately only based on the ZDDs of their common ancestor $X^d$. Note the ZDD construction strategy is the same to the conventional ZDD-based method. Therefore, in our method, we only consider to use the serial pattern $d$-separation to partition a BN.

### 3.2 On-line Inference Time Consumption

For any given BN, it can be recursively partitioned into several components to further reduce ZDD size until no serial pattern $d$-separations are found. One main problem is that as the number of components increases, time for on-line inference may increase unacceptably.

As the partitioning shown in Equation (17), to perform the summing over operations $\sum_{X^{(1)}} P(X^{(1)}, X^d)$, marginal probabilities $P(X^d)$ need to be calculated first. After compiling them into ZDDs, this calculation can be implemented by evaluating all the ZDD paths containing variables corresponding to $P(x_i^{(2)} = \sum_{X^d} P(x_i^{(2)} \mid X^d)$ for all $x_i^{(2)} \in \text{Dom}(X^d)$. These evaluations may lead to considerable time consumption which is exponential to $|\text{Dom}(X^d)|$, the number of instantiations over vertexes in $\text{Dom}(X^d)$. Thus partitioning a BN with a $d$-separation consisting of too many vertexes will lead to a blow up in the number of instantiations in the $d$-separation. As a result, the best choice of a serial pattern $d$-separation would be the vertex sets that can result in a partitioning with minimum $|\text{Dom}(X^d)|$ and minimum ZDD size. Find a set of such $d$-separations is a very complex task.

In this paper, to trade-off time consumption for compiling and inference, we decided to use only the set of all $d$-separations consisting of one vertex because they are highly effective and easily extracted without any complicated evaluation. We use one-vertex $d$-separation to denote the $d$-separation consisting of only one vertex. To enumerate all one-vertex $d$-separations, we use Tarjan’s vertex-cut algorithm which can be carried out in linear time in the number of BN vertexes. Using all one-vertex $d$-separations,
a BN in partitioned into several conditionally independent components. The experiments will show that most of BNs contain one-vertex $d$-separations. Partitioning BNs using all the one-vertex $d$-separations can improve both compilation and inference a lot and even works better than partitioning with two-vertex $d$-separations, $d$-separations consisting of two vertexes.

### 3.3 Enumerating one-vertex $d$-separations

Tarjan’s vertex-cut algorithm [Tarjan 74] is known as one of the most famous algorithms used in an undirected graph to enumerate all cut vertexes (a vertex such that deleting edges connected to the vertex can decompose an undirected graph into two or more components). The algorithm is implemented in time linear in the number of vertexes in the given graph based on the idea of Depth First Search (DFS). In our method, we use this algorithm on an equivalent undirected graph of the given BN to get a set of cut vertexes that contains all one-vertex $d$-separation candidates. Then we pick out the serial pattern $d$-separations and use all of them to partition a BN.

To use Tarjan’s vertex-cut algorithm, first we need to transfer a BN into its equivalent undirected form which is known as Moral Graph.

**Definition 2** [Koller 09] The Moral Graph $M(V)$ of a Bayesian Network $G$ is the undirected graph over $V$ that contains an undirected edge between vertexes $X_i$ and $X_j$ if:

- there is a directed edge between them (in either direction) or
- $X_i$ and $X_j$ are both parents of the same vertex.

Given a BN, first it is moralized into an $M(V)$. We run Tarjan’s vertex-cut algorithm once on $M(V)$ and get a vertex set $X^*$ as shown in Algorithm 1 and 2. Thus, $\forall X_i \in X^*$, deleting edges connected to $X_i$ separates $M(V)$ into several components.

**Theorem 1** [Koller 09] Let $X^{(1)}, X^{(2)}, X^{(3)}$ be three disjoint vertex sets in a BN and $V = X^{(1)} \cup X^{(2)} \cup X^{(3)}$. We say that $X^{(1)}$ is independent to $X^{(2)}$ given $X^{(3)}$ if $X^{(1)}$ is separated from $X^{(2)}$ by $X^{(3)}$ in $M(V)$.

According to the Theorem 1, vertexes in different components decomposed by $X_i \in X^*$ are independent to each other. Now we know that every $X_i \in X^*$, $\{X_i\}$ can be treated as a separation which decomposes the BN into several conditionally independent components. Next we check whether $\{X_i\}$ and these components satisfies serial pattern or not.

For every decomposition by $\{X_i\}$, vertex sets $X^{(1)}$ (where $X^{(1)} \cap \Pi_i \neq \emptyset$) are treated as the upstream component, and all other vertexes are treated as the downstream component $X^{(2)}$. Then, $\{X_i\}$ is a serial pattern $d$-separation $X^d$ such that:

- There are no streams flowing up to the upstream components from $X_i$: $\nexists X_j \in X^{(1)}$, $X_i \in \Pi_j$ or
- There are no streams flowing up to $X_i$ from downstream components: $\nexists X_j \in X^{(2)}$, $X_j \in \Pi_i$.

Using above checking rules, we can pick out all the one-vertex serial pattern $d$-separations from the separation vertex set $X^*$. Then, we use them all to recursively partition a given BN into several components. For the example in Figure 7(a), its corresponding moral graph is shown in (b). Vertexes $X_3$ and $X_5$ are found as cut vertexes using Tarjan’s vertex-cut algorithm. Through the checking rules, they are both confirmed as serial pattern $d$-separations. Then we use this two vertexes to recursively partition a BN into three components as shown in Figure 7(c). Note that no matter in what order we use these one-vertex $d$-separations, we can always get the same partitioning. If we iteratively partition this BN by choosing $X_3$ first and then choosing $X_5$, we get the same partitioning as using $X_5$ first. However, for vertex sets $\{X_3, X_4\}$ and $\{X_4, X_5\}$ in Figure 7 which are both two-vertex serial pattern $d$-separations, if we consider two-vertex $d$-separations, we have to choose one using some complicated evaluation since partitioning with either of them leads to different results. This is another advantage of restricting serial pattern $d$-separations to one-vertex.

**Algorithm 1** Tarjan’s vertex-cut algorithm

**Input:**

$M(V)$;

**Output:**

set of cut vertexes $X^*$;

1: $X^* = \emptyset$;
2: for $X_i \in V$ do
3: $\text{parent}[X_i]=\text{NULL}; \text{cut}[X_i]=\text{FALSE};$
4: $\text{visit}[X_i]=-1;$
5: end for
6: $\text{time}=0;$
7: for $X_i \in V$ do
8: if $\text{visit}[X_i]==-1$ then
9: $\text{DFS}_\text{Visit}(X_i);$
10: end if
11: end for
12: for $X_i \in V$ do
13: if $\text{cut}[X_i]==\text{TRUE}$ then
14: $X^* = X^* \cup X_i;$
15: end if
16: end for
17: return $X^*$
4. Separate Compiling and On-line Inference

In this chapter, we introduce how to compile these components into MLFs so that we can construct ZDDs and introduce how to use ZDDs to execute exact inference.

4.1 MLFs for Independent Components

While partitioning a BN into $G^1, G^2$ with d-separation $X^d$ as shown in Figure 6(a), we take the same idea with the conventional ZDD-based method that constructs a ZDD for one vertex depending on its parents’ ZDDs. Thus, for the vertexes in $G^1$, MLFs for them are the same defined as in equations (7), (8) and (9). The difference is the MLFs for the vertexes in $G^2$. For the vertexer who does not have $X^d$ as its parent, the MLF for it is the same as defined in the conventional ZDD-based method. But for vertexes which have $X^d$ as their parents, if $X_i \in V^2$ only has $X^d$ as its parents, the MLF for $X_i$ is:

\[
\text{MLF}_{X_i} = \sum_{k:x_i,k \in \text{Dom}(X_i)} \text{MLF}_{x_i,k}, \quad \text{(19)}
\]

\[
\text{MLF}_{x_i,k} = \sum_{j:x_i^d \in \text{Dom}(X^d)} \lambda_i, x_{i,j}, \text{MLF}_{x_i^d}, \quad \text{(20)}
\]

where $X_i \in V^2$ and $\Pi_i = X^d$.

If $X_i \in V^2$ also has other vertexes as its parents, then its MLF is given by:

\[
\text{MLF}_{X_i} = \sum_{k:x_i,k \in \text{Dom}(X_i)} \text{MLF}_{x_i,k} \quad \text{(21)}
\]

\[
\text{MLF}_{x_i,k} = \sum_{j:x_i^d \in \text{Dom}(X^d)} \lambda_i, x_{i,j}, \text{MLF}_{x_i^d}, \prod_{l \in \text{Dom}(X^d)} \text{MLF}_{x_i^d}, \quad \text{(22)}
\]

where $X_i \in V^2$, $X^d \subseteq \Pi_i$ and $a_i^d$ is consistent with $\pi_{i,j}$.

For the MLF of $X^d = \{X_i\} \subset V^2$, we introduce $\beta_i \equiv \{\beta_{i,j}\}$ as separator variables that map each instantiation of $X^d$ into a real number. We define MLF $X^d$ as:

\[
\text{MLF}_{X^d} = \sum_{l} \text{MLF}_{x_i^d}, \quad \text{MLF}_{x_i^d} = \lambda_i, \beta_{i,l}, \text{where } X^d = \{X_i\}. \quad \text{(23)}
\]

Note that $\beta_{i,l}$ dynamically changes and takes the values calculated from upstream components according to different queries:

\[
\beta_{i,l} \leftarrow \text{MLF}_{x_i,l}, \quad \text{where } \{X_i\} = X^d \quad \text{(24)}
\]

Given a BN, after partitioning it with a set of $X^d$, we generate ZDDs $Z(X_i)$ for every $X_i \in V$ in the BN. In addition, we generate ZDDs $Z(X^d)$ for every $X^d$.

For the example in Figure 7, suppose every vertex contains two values. For the vertexes $\{X_1, X_2, X_3\}$ in the first component, we have MLFs in the same way as that of conventional ZDD-based method. Then for the vertexes in the second component, MLF $X_i$ is generated the same to conventional ZDD-based method. The MLF for d-separation vertex $X_3$ is:

\[
\text{MLF}_{x_{3,1}^d} = \beta_{3,1} \lambda_{3,1}, \quad \text{MLF}_{x_{3,2}^d} = \beta_{3,2} \lambda_{3,2}. \quad \text{(25)}
\]

Then,

\[
\text{MLF}_{X_3} = \lambda_{3,1}(\theta_{3,1,1} \text{MLF}_{x_{3,1}^d} + \theta_{3,1,2} \text{MLF}_{x_{3,2}^d} - \lambda_{3,2}) + \lambda_{3,2}(\theta_{3,2,1} \text{MLF}_{x_{3,1}^d} + \theta_{3,2,2} \text{MLF}_{x_{3,2}^d}). \quad \text{(26)}
\]

Similarly, for the third component, MLFs for $X_5, X_6$ and $X_7$ are:

\[
\text{MLF}_{x_{6,1}^d} = \beta_{5,1} \lambda_{5,1}, \quad \text{MLF}_{x_{6,2}^d} = \beta_{5,2} \lambda_{5,2}. \quad \text{(27)}
\]

\[
\text{MLF}_{X_6} = \lambda_{6,1}(\theta_{6,1,1} \text{MLF}_{x_{6,1}^d} + \theta_{6,1,2} \text{MLF}_{x_{6,2}^d}) + \lambda_{6,2}(\theta_{6,2,1} \text{MLF}_{x_{6,1}^d} + \theta_{6,2,2} \text{MLF}_{x_{6,2}^d}). \quad \text{(28)}
\]

\[
\text{MLF}_{X_7} = \lambda_{7,1}(\theta_{7,1,1} \text{MLF}_{x_{7,1}^d} + \theta_{7,1,2} \text{MLF}_{x_{7,2}^d}) + \lambda_{7,2}(\theta_{7,2,1} \text{MLF}_{x_{7,1}^d} + \theta_{7,2,2} \text{MLF}_{x_{7,2}^d}). \quad \text{(29)}
\]
4.2 Exact Inference with ZDDs

Usually, the task of exact inference is to calculate the probability of some instances \( x_i^{(1)} \) given some observations \( x_k^{(2)} \): \( P(x_i^{(1)} | x_k^{(2)}) \). According to the definition of conditional probability, we have:

\[
P(x_i^{(1)} | x_k^{(2)}) = \frac{P(x_i^{(1)}, x_k^{(2)})}{P(x_k^{(2)})}. \tag{30}
\]

If we can efficiently calculate \( P(x_i^{(1)}, x_k^{(2)}) \) and \( P(x_k^{(2)}) \), the probability of this query can be easily obtained. If the variables to be inferred and variables observed are in the same component, the probability can be calculated the same as the conventional ZDD-based method. The problem is when these variables are in different components, we need to take efforts on separation variables. Whenever we visit a ZDD node representing a separate variable \( \beta \), we get its value by tracing corresponding ZDDs to collect the information passed from other components.

Let’s considering the following example of calculating \( P(x_{2,1}, x_{5,1}) \) with the BN in Figure 7(c). First, we have:

\[
MLF_{x_{5,1}} = \lambda_{3,1} \lambda_{4,1} \lambda_{5,1} \beta_{3,1} \theta_{4,1,1} \theta_{5,1,1} + \lambda_{3,2} \lambda_{4,1} \lambda_{5,1} \beta_{3,2} \theta_{4,1,1} \theta_{5,2,1} + \lambda_{3,2} \lambda_{4,2} \lambda_{5,1} \beta_{3,2} \theta_{4,2,1} \theta_{5,4,1}. \tag{31}
\]

By setting \( \lambda \) to 1, we get:

\[
MLF_{x_{5,1}} = \beta_{3,1} \theta_{4,1,1} \theta_{5,1,1} + \beta_{3,2} \theta_{4,1,1} \theta_{5,2,1} + \beta_{3,2} \theta_{4,1,2} \theta_{5,4,1}. \tag{32}
\]

For parameter variable \( \theta \)s, they take values as in CPT. For separator variable \( \beta_{3,1} \) and \( \beta_{3,2} \), we trace the upstream ZDD to get the information. Since we already have:

\[
MLF_{X_2} = MLF_{x_{2,1}} + MLF_{x_{2,2}} = \lambda_{2,1} \theta_{2,1,1} + \lambda_{2,2} \theta_{2,1,2}. \tag{33}
\]

\[
MLF_{X_3} = MLF_{x_{3,1}} + MLF_{x_{3,2}} = \lambda_{1,1} \lambda_{2,1} \lambda_{3,1} \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,1} + \lambda_{1,1} \lambda_{2,1} \lambda_{3,2} \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,2} + \lambda_{1,2} \lambda_{2,2} \lambda_{3,2} \theta_{1,1,2} \theta_{2,1,2} \theta_{3,4,2}. \tag{34}
\]

According to the query, we generate:

\[
MLF_{x_{2,1}} MLF_{x_{3,1}} = \lambda_{1,1} \lambda_{2,1} \lambda_{3,1} \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,1} + \lambda_{1,2} \lambda_{2,1} \lambda_{3,1} \theta_{1,1,2} \theta_{2,1,1} \theta_{3,2,1}. \tag{35}
\]

\[
MLF_{x_{2,1}} MLF_{x_{3,2}} = \lambda_{1,1} \lambda_{2,1} \lambda_{3,2} \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,2} + \lambda_{1,2} \lambda_{2,1} \lambda_{3,2} \theta_{1,1,2} \theta_{2,1,2} \theta_{3,2,2}. \tag{36}
\]

By setting \( \lambda \)s to 1, we can get:

\[
\beta_{3,1} \leftarrow MLF_{x_{2,1}} MLF_{x_{3,1}} = \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,1} + \theta_{1,1,2} \theta_{2,1,1} \theta_{3,2,1}. \tag{37}
\]

\[
\beta_{3,2} \leftarrow MLF_{x_{2,1}} MLF_{x_{3,2}} = \theta_{1,1,1} \theta_{2,1,1} \theta_{3,1,2} + \theta_{1,1,2} \theta_{2,1,2} \theta_{3,2,2}. \tag{38}
\]

Substituting \( \beta_{3,1}, \beta_{3,2} \) into MLF\(_{x_{5,1}}\), we can get \( P(x_{2,1}, x_{5,1}) \). By using the same multiplication algorithm as in the conventional ZDD-based method, no term can contain the same variable more than once. Contradicting terms are automatically eliminated. We can always generate ZDDs containing variables only related to the query so that unnecessary calculations are avoided. An important point is that our idea of partitioning is independent to queries. Once we partition a BN with \( d \)-separations and generate ZDDs, we can use them to calculate probabilities for any queries.

One may question that whether this calculation is efficient enough for calculating the conditional probability such as \( P(x_{2,1} | x_{5,1}) \) since we have to calculate \( P(x_{5,1}) \).
and \( P(x_{2,1}, x_{5,1}) \) first to get the final results. Note that
\[
P(x_{2,1}, x_{5,1}) = \sum_{X_1, X_2} P(X_1, X_2 = x_{2,1}, X_3, X_4, X_5 = x_{5,1}),
\]
(39)
\[
P(x_{5,1}) = \sum_{X_1, X_2} P(X_1, X_2 = x_{2,1}, X_3, X_4, X_5 = x_{5,1})
+ \sum_{X_1, X_2} P(X_1, X_2 = x_{2,2}, X_3, X_4, X_5 = x_{5,1}).
\]
(40)

After we calculate \( P(x_{2,1}, x_{5,1}) \), we only need to calculate the second part in formula (40) for \( P(x_{5,1}) \). Repetitive calculations can be avoided using the cache memory technique in ZDDs.

5. Experiments and Results

5.1 Overview

<table>
<thead>
<tr>
<th>Table 1</th>
<th>BN specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dataset</td>
</tr>
<tr>
<td></td>
<td>ALARM</td>
</tr>
<tr>
<td></td>
<td>WIN95PTS</td>
</tr>
<tr>
<td></td>
<td>HEPAR2</td>
</tr>
<tr>
<td></td>
<td>HAILFINDER</td>
</tr>
<tr>
<td></td>
<td>PATHFINDER</td>
</tr>
<tr>
<td></td>
<td>INSURANCE</td>
</tr>
<tr>
<td></td>
<td>MILDEW</td>
</tr>
<tr>
<td></td>
<td>WATER</td>
</tr>
<tr>
<td></td>
<td>PIGS</td>
</tr>
<tr>
<td></td>
<td>BARLEY</td>
</tr>
<tr>
<td></td>
<td>DIABETES</td>
</tr>
</tbody>
</table>

We conducted our experiments using the benchmark networks[Hebrew University] and the ZDD package implemented by Minato [Minato 01] on an intel Core i7-2700k CPU @ 3.50GHz x 8PC with Ubuntu 16.04LTS and 31.4GiB of main memory. The network specifications such as BN name, the number of vertexes, indicators, and parameters are shown in Table 1.

The results of our proposed method comparing with the conventional ZDD-method is shown in Table 2. For a given BN, first, we find one-vertex \( d \)-separations and use them all to partition the BN. Time for finding \( d \)-separations and partitioning are together shown in column “Time(d-sep)(ms)”. The number of \( d \)-separations we use are shown in the last column before the bracket. In the off-line compilation, we generate ZDDs for this BN. Then, we calculate values of separator variables \( \beta \_s \) by setting all \( \lambda \_s \) to 1 in advance. Cache memories for these calculations would help to improve on-line inference. The ZDD size is shown in the first column. Time of generating ZDDs and calculating \( \beta \_s \) are shown as “Compiling(ms)” represented in the second column. For the on-line inference, we conduct two kinds of exact inference. First is to calculate the marginal probability of every vertex in the BN. Time for the this inference is shown in column “Mar.(ms)”. The second is to infer one hundred of instantiations. we randomly selected hundred pairs of BN vertexes and randomly instantiate them. Then we calculate marginal probabilities over these instantiations. The total time of computation for these instantiations are presented in the column named “Arb.(ms)”. Time limitation in our experiments is set within 30 minutes.

To show the validity of partitioning with one-vertex \( d \)-separations, we also conducted the experiments of partitioning a BN using one- or two-vertex \( d \)-separations in Table 3.

5.2 Results and Discussion

According to Table 2, most of the BNs contain one-vertex serial pattern \( d \)-separations. Our method performs quite well than the conventional ZDD-based method on condensing ZDD size for networks such as ALARM, HEPAR2 and PIGS, which present obviously layer-wised structures. Also, both time of compilation and inference are reduced significantly on these networks. But network WATER does not contain one-vertex serial pattern \( d \)-separations. Vertexes in this network are structured just in 4 layers and closely connect to each other. For networks such as MIDLEW and PATHFINDER, there are just few one-vertex serial pattern \( d \)-separations. Some of the \( d \)-separations are located in the corner of the networks so that partitioning with these \( d \)-separations does not bring much improvement while compiling them into ZDDs. Also, the inference time for hundred vertex pairs shown in column “Arb(ms)” are largely reduced through reusing the results of probabilities over \( d \)-separations based on the cache memory technique. An example of partitioning ALARM is shown in Figure 8.

One may question the validity of partitioning only with one-vertex \( d \)-separations because intuitively, \( d \)-separations which can decompose a BN into two balanced components are considered to be more likely to generate small ZDDs. However, if the \( d \)-separation consists of too many vertexes, inference time after partitioning may be unacceptable because inference time greatly depends on \( |\text{Dom}(X^d)| \) which grows exponentially with the number of vertexes in \( X^d \) (suppose every vertex is binary valued). By restricting \( d \)-separation to one-vertex, partitioning with all one-vertex \( d \)-separations is always highly effective in both ZDD size and inference efficiency. In the last column in Table 2, we list the different numbers of component sizes partitioned by the one-vertex \( d \)-separations in the bracket. For the
### Table 2 Experiment results compared with conventional ZDD-based method

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Conventional ZDD-based Method [Minato 07]</th>
<th>Proposed Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALARM</td>
<td>34,299</td>
<td>57</td>
</tr>
<tr>
<td>WIN95PTS</td>
<td>26,477</td>
<td>104</td>
</tr>
<tr>
<td>HEPAR2</td>
<td>51,000</td>
<td>126</td>
</tr>
<tr>
<td>HAILFINER</td>
<td>294,608</td>
<td>467</td>
</tr>
<tr>
<td>PATHFINDER</td>
<td>31,549</td>
<td>2,633</td>
</tr>
<tr>
<td>MILDEW</td>
<td>15,310,511</td>
<td>664,097</td>
</tr>
<tr>
<td>WATER</td>
<td>25,629</td>
<td>874</td>
</tr>
<tr>
<td>PIGS</td>
<td>73,715</td>
<td>517</td>
</tr>
<tr>
<td>BARLEY</td>
<td>time out</td>
<td>time out</td>
</tr>
<tr>
<td>DIABETES</td>
<td>time out</td>
<td>time out</td>
</tr>
<tr>
<td>WIN95PTS</td>
<td>24,460</td>
<td>84</td>
</tr>
<tr>
<td>(Problem1)</td>
<td>1,091,779</td>
<td>4,360</td>
</tr>
</tbody>
</table>

### Table 3 Results of partitioning with heuristically chosen d-separations consisting of one or two vertexes

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Partitioning using all possible one- and two-vertex d-separations</th>
<th>Partitioning with the best greedy selection from one and two-vertex d-separations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZDD size</td>
<td>Compiling (ms)</td>
</tr>
<tr>
<td>ALARM</td>
<td>1,304</td>
<td>10</td>
</tr>
<tr>
<td>WIN95PTS</td>
<td>6,812</td>
<td>55</td>
</tr>
<tr>
<td>HEPAR2</td>
<td>5,544</td>
<td>44</td>
</tr>
<tr>
<td>HAILFINER</td>
<td>29,756</td>
<td>84</td>
</tr>
<tr>
<td>PATHFINDER</td>
<td>31,553</td>
<td>2,621</td>
</tr>
<tr>
<td>MILDEW</td>
<td>7,440,861</td>
<td>740,035</td>
</tr>
<tr>
<td>WATER</td>
<td>25,670</td>
<td>821</td>
</tr>
<tr>
<td>PIGS</td>
<td>13,829</td>
<td>487</td>
</tr>
<tr>
<td>DIABETES</td>
<td>time out</td>
<td>time out</td>
</tr>
<tr>
<td>BARLEY</td>
<td>time out</td>
<td>time out</td>
</tr>
</tbody>
</table>

### Table 4 Results for jointree algorithm

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Min-fill Triangulation</th>
<th>Min-degree Triangulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cluster</td>
<td>Separators (maximum)</td>
</tr>
<tr>
<td>ALARM</td>
<td>27</td>
<td>36</td>
</tr>
<tr>
<td>WIN95PTS</td>
<td>50</td>
<td>128</td>
</tr>
<tr>
<td>HEPAR2</td>
<td>58</td>
<td>96</td>
</tr>
<tr>
<td>HAILFINER</td>
<td>43</td>
<td>297</td>
</tr>
<tr>
<td>PATHFINDER</td>
<td>91</td>
<td>8,064</td>
</tr>
<tr>
<td>MILDEW</td>
<td>29</td>
<td>28,800</td>
</tr>
<tr>
<td>WATER</td>
<td>19</td>
<td>110,592</td>
</tr>
<tr>
<td>PIGS</td>
<td>368</td>
<td>59,049</td>
</tr>
<tr>
<td>BARLEY</td>
<td>36</td>
<td>1,125,600</td>
</tr>
<tr>
<td>DIABETES</td>
<td>335</td>
<td>14,025</td>
</tr>
</tbody>
</table>
Table 5  Results for CNF based Method

<table>
<thead>
<tr>
<th>Dataset</th>
<th>AC Nodes</th>
<th>AC Edges</th>
<th>Compiling(ms)</th>
<th>Mar. (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALARM</td>
<td>1,570</td>
<td>2,848</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>WIN95PTS</td>
<td>3,004</td>
<td>5,638</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>HEPAR2</td>
<td>7,697</td>
<td>11,966</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>HAILFINDER</td>
<td>8,594</td>
<td>16,532</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>PATHFINDER</td>
<td>17,825</td>
<td>33,786</td>
<td>35</td>
<td>9</td>
</tr>
<tr>
<td>MILDEW</td>
<td>1,118,179</td>
<td>2,219,244</td>
<td>241</td>
<td>53</td>
</tr>
<tr>
<td>WATER</td>
<td>28,741</td>
<td>56,496</td>
<td>54</td>
<td>12</td>
</tr>
<tr>
<td>PIGS</td>
<td>625,752</td>
<td>1,248,854</td>
<td>99</td>
<td>43</td>
</tr>
<tr>
<td>BARLEY</td>
<td>18,196,115</td>
<td>36,348,928</td>
<td>1,448</td>
<td>489,891</td>
</tr>
<tr>
<td>DIABETES</td>
<td>7,655,537</td>
<td>15,300,530</td>
<td>828</td>
<td>229</td>
</tr>
</tbody>
</table>

example of ALARM in Figure 8, it is partitioned into 6 components with size 27, 5, 3, 3, 2, 2 respectively, we add (27,5,3,2) in the last column in Table 2. Note for networks of ALARM, WIN95PTS, HEPAR2, HAILFINDER, and PIGS, although they are not partitioned into very balanced components, by cutting off some small components using one-vertex $d$-separations, ZDD size are still reduced a lot so that inference time is also improved. Also, for networks of MILDEW and PATHFINDER which are partitioned into apparently unbalanced large and small components, the efficiency of compilation and inference is not badly affected a lot because the increasing of ZDD nodes are just for compiling the separator variables whose number are controlled by limiting the $d$-separations to one-vertex.

To present the validity of partitioning with all one-vertex $d$-separations, we also conducted the experiments of partitioning with two-vertex $d$-separations shown in Table 3. First all one- and two-vertex serial pattern $d$-separations are enumerated using a complete search approach which costs time about $O(|V|^3)$. Then a serial pattern $d$-separation is chosen as the suitable one such that partitioning with this $d$-separation could give rise to the largest reductions of the number of MLF items, which is an approximation of ZDD size. Using this $d$-separation, the given BN is partitioned into two components and separately compiled into ZDDs. We repeated this procedure for these components until no suitable $d$-separations are found. We generated ZDDs and performed the inference for every iteration to see how the ZDD size and inference time change as the number of components increases. The first four columns in Table 3 are results for iteratively partitioning BNs using all possible one- and two-vertex $d$-separations until no suitable $d$-separations are found. According to results, the ZDD size and time for compilation are largely reduced through partitioning BNs also with two-vertex $d$-separations, which is more likely to partition a BN into balanced components than one-vertex $d$-separations. However, inference time may increase incredibly such as HAILFINDER, MILDEW and PIGS.

What is more, we also present the results that we do not partitioning a BN to the end until no suitable $d$-separations are found, but stop the partitioning at the point that the inference is the fastest. These results are shown in the last four column in Table 3. Though we stop the partitioning at a suitable stage, ZDD size and inference time may be
reduced for networks such as WIN95PTS and HEPAR2. But we have to generate ZDDs and perform the inference repeatedly to the end to choose the best result. Time consumption for such work is unacceptable. Also partitioning on network such as MILDEW still performs rather badly. It is worthy of finding a good measurement of choosing suitable \(d\)-separation and precise trade-off between time for compiling and inference as the future work. At the present stage, our idea of using only one-vertex \(d\)-separations proposed in this paper is a simple and easy-to-use heuristic method. What is more, it has no risk of extremely reducing the inference efficiency.

One may query what if appropriate one-vertex serial pattern \(d\)-separations do not generally exist in the practical BNs. Note one main advantage of ZDD-based compiling approach is that when we query the probability of a vertex, we can only focus on the subgraph which consists of the concerned vertex and all its ancestors. Therefore, we do not have to find the \(d\)-separations that partitions the whole BN but find the \(d\)-separation that can partition the subgraph. The subgraph may contain suitable one-vertex \(d\)-separations. For the example of subnetwork “WIN95PTS(Problem1)” which is formed with vertex “Problem1” and all its ancestors (36 vertexes in total). Partitioning in this subnetwork brings significant improvement in both ZDD size and inference time than conventional ZDD-based method as shown in Table 2. Also, for the subnetwork “BARLEY(ngtilg)”, though we could not generate ZDDs for the whole BARLEY, we still can query some of the vertexes by constructing the subnetwork.

6. Related Work

The most famous method to decompose a BN into another structure is the jointree algorithm. It decomposes a BN into a tree structure and message passing algorithm for exact inference is carried out based on this tree. The treewidth of a jointree dominates the complexity of the exact inference. Given a jointree with treewidth \(w\), node marginals of BNs can be computed in time and space exponential in \(w\). To generate a jointree, one needs to triangulate the moral graph of the given BN. Finding the best order to triangulate a moral graph will lead to a jointree with minimum treewidth. However, this is known as a NP hard problem. Constructing a jointree with minimal width is a hot topic for jointree based approaches [Li 15, Li 17, Ottosen 12]. The Minimum fill-in method (min-fill) and minimum degree method (min-degree) are known as the most two famous heuristic methods for the triangulation to construct jointrees [Darwiche 09]. The min-fill is to eliminate the variable that leads to adding the smallest number of fill-in edges. The min-degree is to eliminate the variable that has the smallest number of neighbors in the graph. Different with the jointree algorithms, our ZDD-based method mainly aims at exploiting local structures in the BNs. It is believed that exploiting the local structure can speed up inference to the point of beating the treewidth barrier [Darwiche 10]. Through the techniques of node-sharing and cache memory in ZDDs, local structures, in the form of variables with same values and local computations [Gao 17] are efficiently exploited. Instead of treewidth, the ZDD size is the main factor that affects the inference efficiency. We present the conventional jointree based method results using min-fill and min-degree in Table 4. According to this table, the on-line inference using ZDD-based method works much better than jointree based methods as long as we are able to generate ZDDs for a BN. Therefore, efficiently generating ZDDs for BNs is quite valuable to be taken into account.

The method in [Gao 17] proposed to separately compile a BN by compiling a given jointree into ZDDs and perform the message passing algorithm on ZDDs. In their method, they introduce message variables into ZDDs to receive and pass messages. However, straightforwardly compiling a jointree into ZDDs results in a large number of components and message variables so that the time consumption for compiling and inference both are unacceptable. In our method, through restricting the \(d\)-separations size to one-vertex, we avoid partitioning a BN into too many components and using too many message variables. Thus, the inference efficiency is guaranteed.

Another popular method to compile a BN using symbolic logics is the CNF-based compiling method [Darwiche 04]. In their method, BNs is encoded into Conjunctive Normal Forms (CNFs) which then can be factored and compactly represented by arithmetic circuits. Evaluating and Differentiating the arithmetic circuits solves the exact inference efficiently. Their method also provides expressive frameworks for exploiting local structure and known as one of the most efficient methods that compiles BN with Decision Diagrams. Our method does not use the CNF representation but directly translates a BN into a set of factored MLFs using ZDDs. Table 5 is the results of their method using the same dataset in Table 1. For the networks ALARM and HEPAR2, the ZDD-based method are competitive to theirs through partitioning. Also, our idea of partitioning BNs is independent of data structures so that we hope it would also fit their method.
7. Conclusion and Future Work

We proposed a method of divide-and-conquer that partitions BNs into conditionally independent components using one-vertex serial pattern $d$-separations and separately compile these components. Through the partitioning, we get much smaller ZDDs and largely reduce time for compilation than the conventional ZDD-based method if there exists suitable one-vertex $d$-separations in given BNs. Through our experiments, we know that we can partition a given BN with all one-vertex $d$-separations without worrying about reducing the efficiency of compiling and inference.

As a future work, we hope to use $d$-separations consisting of more than one vertex to partition a BN and inspecting the trade-off between compiling and inference to further improve the ZDD-based method for BN inference.

Acknowledgments

This work is partly supported by JSPS KAKENHI(S) 15h05711.

References


[Cooper 88] Cooper, G. F.: An algorithm for computing probabilistic propositions, Knowledge Systems Lab Reports, Computer Science Department, Stanford University, KSL-87-21 (1988)


Authors’ Profile

Shan Gao

is currently a Ph.D candidate at the Hokkaido University. She received a M.E. degree in 2015 from Hokkaido University and a B.E. degree from Dalian University of Technology, China in 2012. Her research interests include efficient inference algorithms for probabilistic modeling using discrete structures. She was a research assistant for IST ERATO MINATO Discrete Structure Manipulation System.
Project from 2013 to 2016. She is now a research assistant for JSPS KAKENHI(S) Project from 2016.

**Masakazu Ishihata (Member)**
is a temporarily assigned assistant professor at the Graduate School of Information Science and Technology, Hokkaido University. He received the associate degree of Information Science from Ishikawa National College of Technology in 2006, and the bachelor, master, and doctor degrees in Computer Science from Tokyo Institute of Technology in 2008, 2010, and 2013, respectively. He was a Research Fellow for Young Scientists (DC1) of Japan Society for the Promotion of Science (JSPS) from 2010 to 2013. He worked for NTT Communication Science Laboratories from 2013 to 2016. He temporally moved to Hokkaido University as an assistant professor in 2016. His research interests include applications of discrete structures to efficient probabilistic modeling, combinatorial optimization, and so on.

**Shin-ichi Minato (Member)**
is a Professor at the Graduate School of Information Science and Technology, Hokkaido University. He received the B.E., M.E., and D.E. degrees in Information Science from Kyoto University in 1988, 1990, and 1995, respectively. He worked for NTT Laboratories from 1990 until 2004. He was a Visiting Scholar at the Computer Science Department of Stanford University in 1997. He joined Hokkaido University as an Associate Professor in 2004, and has been a Professor since 2010. His research interests include efficient representations and manipulation algorithms for large-scale data of discrete structures. He served a Research Director of JST ERATO MINATO Discrete Structure Manipulation System Project from 2009 to 2016, and now he is leading JSPS KAKENHI(S) Project until 2020. He is a senior member of IEICE and IPSJ, and is a member of IEEE.